20 | 日志处理（上）：如何设计日志包并记录日志？

2021-07-10 孔令飞

《Go 语言项目开发实战》 课程介绍 >

讲述：孔令飞
时长 24:44 大小 22.66M

你好，我是孔令飞，接下来的两讲，我们来聊聊如何设计和开发日志包。

在做 Go 项目开发时，除了处理错误之外，我们必须要做的另外一件事是记录日志。通过记录日志，可以完成一些基本功能，比如开发、测试期间的 Debug，故障排除，数据分析，监控告警，以及记录发生的事件等。

要实现这些功能，首先我们需要一个优秀的日志包。另外，我还发现不少 Go 项目开发者记录日志很随意，输出的日志并不能有效定位到问题。所以，我们还需要知道怎么更好地记录日志，这就需要一个日志记录规范。

有了优秀的日志包和日志记录规范，我们就能很快地定位到问题，获取足够的信息，并完成后期的数据分析和监控告警，也可以很方便地进行调试了。这一讲，我就来详细介绍

下，如何设计日志包和日志记录规范。

首先，我们来看下如何设计日志包。

如何设计日志包

目前，虽然有很多优秀的开源日志包可供我们选择，但在一个大型系统中，这些开源日志包很可能无法满足我们定制化的需求，这时候我们就需要自己开发日志包。

这些日志包可能是基于某个，或某几个开源的日志包改造而来，也可能是全新开发的日志包。那么在开发日志包时，我们需要实现哪些功能，又如何实现呢？接下来，我们就来详细聊聊。

先来看下日志包需要具备哪些功能。根据功能的重要性，我将日志包需要实现的功能分为**基础功能**、**高级功能**和**可选功能**。基础功能是一个日志包必须要具备的功能；高级功能、可选功能都是在特定场景下可增加的功能。我们先来说基础功能。

基础功能

基础功能，是优秀日志包必备的功能，能够满足绝大部分的使用场景，适合一些中小型的项目。一个日志包应该具备以下 4 个基础功能。

1. 支持基本的日志信息

日志包需要支持基本的日志信息，包括时间戳、文件名、行号、日志级别和日志信息。

时间戳可以记录日志发生的时间。在定位问题时，我们往往需要根据时间戳，来复原请求过程，核对相同时间戳下的上下文，从而定位出问题。

文件名和行号，可以使我们更快速定位到打印日志的位置，找到问题代码。一个日志库如果不支持文件名和行号，排查故障就会变得非常困难，基本只能靠 grep 和记忆来定位代码。对于企业级的服务，需要保证服务在故障后能够快速恢复，恢复的时间越久，造成的损失就越大，影响就越大。这就要求研发人员能够快速定位并解决问题。通过文件名和行号，我们可以精准定位到问题代码，尽快地修复问题并恢复服务。

通过日志级别，可以知道日志的错误类型，最通常的用法是：直接过滤出 Error 级别的日志，这样就可以直接定位出问题出错点，然后再结合其他日志定位出出错的原因。如果不支持日志级别，在定位问题时，可能要查看一大堆无用的日志。在大型系统中，一次请求的日志量很多，这会大大延长我们定位问题的时间。

而通过日志信息，我们可以知道错误发生的具体原因。

2. 支持不同的日志级别

不同的日志级别代表不同的日志类型，例如：Error 级别的日志，说明日志是错误类型，在排障时，会首先查看错误级别的日志。Warn 级别日志说明出现异常，但还不至于影响程序运行，如果程序执行的结果不符合预期，则可以参考 Warn 级别的日志，定位出异常所在。Info 级别的日志，可以协助我们 Debug，并记录一些有用的信息，供后期进行分析。

通常一个日志包至少要实现 6 个级别，我给你提供了一张表格，按优先级从低到高排列如下：

日志级别	说明
Debug	Debug级别的日志主要用来提供一些Debug信息，方便我们在开发测试时，定位问题，观察程序的运行是否符合预期等
Info	Info通常是默认的日志级别，该日志级别的日志主要目的是提供一些必要的日志信息，方便在业务出现问题时，结合Error级别的日志，快速地排障
Warn	Warn级别的日志，比Info级别的日志更加重要
Error	Error级别的日志，代表程序执行出错，如果程序正常运行，不应该产生任何Error级别的日志。Error级别的日志可以直接让我们定位到出错的原因，所以在实际的开发中，通常会把Error级别的日志输出到一个独立的文件中，方便出问题时查看
Panic	Panic级别的日志，说明错误比较严重，在输出Panic日志内容后，会调用panic(s)（s为日志内容）使程序Panic，并打印出错误堆栈，方便我们定位问题。如果程序设置了defer…recover，Panic还可以被defer…recover捕获
Fatal	Fatal级别的日志，表示程序遇到了致命的错误，需要退出。在输出日志内容后，调用os.Exit退出程序。例如，对于一个业务API，如果连接数据库失败，则可以直接调用log.Fatal打印错误，并退出程序，因为如果连接数据库失败，那我们就没法做业务的增删改查，程序没法继续执行

有些日志包，例如 logrus，还支持 Trace 日志级别。Trace 级别比 Debug 级别还低，能够打印更细粒度的日志信息。在我看来，Trace 级别不是必须的，你可以根据需要自行选

择。

打印日志时，一个日志调用其实具有两个属性：

输出级别：打印日志时，我们期望日志的输出级别。例如，我们调用 `glog.Info("This is info message")` 打印一条日志，则输出日志级别为 Info。

开关级别：启动应用程序时，期望哪些输出级别的日志被打印。例如，使用 glog 时 `-v=4` ，说明了只有日志级别高于 4 的日志才会被打印。

如果开关级别设置为 `L` ，只有输出级别 `>=L` 时，日志才会被打印。例如，开关级别为 Warn，则只会记录 Warn、Error 、Panic 和 Fatal 级别的日志。具体的输出关系如下图所示：

开关级别 \ 输出级别	Fatal	Panic	Error	Warn	Info	Debug	Trace
Fatal	■						
Panic	■	■					
Error	■	■	■				
Warn	■	■	■	■			
Info	■	■	■	■	■		
Debug	■	■	■	■	■	■	
Trace	■	■	■	■	■	■	■

绿色背景代表可见，当设置开关级别为Trace时，会打印程序中所有级别的日志

3. 支持自定义配置

不同的运行环境，需要不同的日志输出配置，例如：开发测试环境为了能够方便地 Debug，需要设置日志级别为 Debug 级别；现网环境为了提高应用程序的性能，则需要设置日志级别为 Info 级别。又比如，现网环境为了方便日志采集，通常会输出 JSON 格式的日志；开发测试环境为了方便查看日志，会输出 TEXT 格式的日志。

所以，我们的日志包需要能够被配置，还要不同环境采用不同的配置。通过配置，可以在不重新编译代码的情况下，改变记录日志的行为。

4. 支持输出到标准输出和文件

日志总是要被读的，要么输出到标准输出，供开发者实时读取，要么保存到文件，供开发者日后查看。输出到标准输出和保存到文件是一个日志包最基本的功能。

高级功能

除了上面提到的这些基本功能外，在一些大型系统中，通常还会要求日志包具备一些高级功能。这些高级功能可以帮我们更好地记录日志，并实现更丰富的功能，例如日志告警。那么一个日志包可以具备哪些高级功能呢？

1. 支持多种日志格式

日志格式也是我们要考虑的一个点，一个好的日志格式，不仅方便查看日志，还能方便一些日志采集组件采集日志，并对接类似 Elasticsearch 这样的日志搜索引擎。

一个日志包至少需要提供以下两种格式：

TEXT 格式：TEXT 格式的日志具有良好的可读性，可以方便我们在开发联调阶段查看日志，例如：

2020-12-02T01:16:18+08:00 INFO example.go:11 std log

2020-12-02T01:16:18+08:00 DEBUG example.go:13 change std log to debug level

JSON 格式：JSON 格式的日志可以记录更详细的信息，日志中包含一些通用的或自定义的字段，可供日后的查询、分析使用，而且可以很方便地供 filebeat、logstash 这类日志采集工具采集并上报。下面是 JSON 格式的日志：

复制代码

```
{"level":"DEBUG","time":"2020-12-02T01:16:18+08:00","file":"example.go:15","fu
{"level":"INFO","time":"2020-12-02T01:16:18+08:00","file":"example.go:16","fun
```

我建议在开发联调阶段使用 TEXT 格式的日志，在现网环境使用 JSON 格式的日志。一个优秀的日志库，例如 logrus，除了提供基本的输出格式外，还应该允许开发者自定义日志输出格式。

2. 能够按级别分类输出

为了能够快速定位到需要的日志，一个比较好的做法是将日志按级别分类输出，至少错误级别的日志可以输出到独立的文件中。这样，出现问题时，可以直接查找错误文件定位问题。例如，glog 就支持分类输出，如下图所示：

```
[colin@dev log]$ ll
total 12
-rw-rw-r-- 1 colin colin 317 Dec  2 08:11 main.colin.colin.log.ERROR.20201202-081133.24123
-rw-rw-r-- 1 colin colin 589 Dec  2 08:11 main.colin.colin.log.INFO.20201202-081133.24123
-rw-rw-r-- 1 colin colin 456 Dec  2 08:11 main.colin.colin.log.WARNING.20201202-081133.24123
lrwxrwxrwx 1 colin colin  48 Dec  2 08:11 main.ERROR -> main.colin.colin.log.ERROR.20201202-081133.24123
lrwxrwxrwx 1 colin colin  47 Dec  2 08:11 main.INFO -> main.colin.colin.log.INFO.20201202-081133.24123
lrwxrwxrwx 1 colin colin  50 Dec  2 08:11 main.WARNING -> main.colin.colin.log.WARNING.20201202-081133.24123
```

3. 支持结构化日志

结构化日志（Structured Logging），就是使用 JSON 或者其他编码方式使日志结构化，这样可以方便后续使用 Filebeat、Logstash Shipper 等各种工具，对日志进行采集、过滤、分析和查找。就像下面这个案例，使用 zap 进行日志打印：

复制代码

```go
package main

import (
    "time"

    "go.uber.org/zap"
)

func main() {
    logger, _ := zap.NewProduction()
    defer logger.Sync() // flushes buffer, if any
    url := "http://marmotedu.com"
    // 结构化日志打印
    logger.Sugar().Infow("failed to fetch URL", "url", url, "attempt", 3, "bac

    // 非结构化日志打印
    logger.Sugar().Infof("failed to fetch URL: %s", url)
}
```

上述代码输出为：

复制代码

```
{"level":"info","ts":1607303966.9903321,"caller":"zap/structured_log.go:14","m
{"level":"info","ts":1607303966.9904354,"caller":"zap/structured_log.go:17","m
```

4. 支持日志轮转

在一个大型项目中，一天可能会产生几十个 G 的日志。为了防止日志把磁盘空间占满，导致服务器或者程序异常，就需要确保日志大小达到一定量级时，对日志进行切割、压缩，并转存。

如何切割呢？你可以按照日志大小进行切割，也可以按日期切割。日志的切割、压缩和转存功能可以基于 GitHub 上一些优秀的开源包来封装，例如：lumberjack可以支持按大小和日期归档日志，file-rotatelogs支持按小时数进行日志切割。

对于日志轮转功能，其实我不建议在日志包中添加，因为这会增加日志包的复杂度，我更建议的做法是借助其他的工具来实现日志轮转。例如，在 Linux 系统中可以使用 Logrotate 来轮转日志。Logrotate 功能强大，是一个专业的日志轮转工具。

5. 具备 Hook 能力

Hook 能力可以使我们对日志内容进行自定义处理。例如，当某个级别的日志产生时，发送邮件或者调用告警接口进行告警。很多优秀的开源日志包提供了 Hook 能力，例如 logrus 和 zap。

在一个大型系统中，日志告警是非常重要的功能，但更好的实现方式是将告警能力做成旁路功能。通过旁路功能，可以保证日志包功能聚焦、简洁。例如：可以将日志收集到 Elasticsearch，并通过 ElastAlert 进行日志告警。

可选功能

除了基础功能和高级功能外，还有一些功能。这些功能不会影响到日志包的核心功能，但是如果具有这些功能，会使日志包更加易用。比如下面的这三个功能。

1. 支持颜色输出

在开发、测试时开启颜色输出，不同级别的日志会被不同颜色标识，这样我们可以很轻松地发现一些 Error、Warn 级别的日志，方便开发调试。发布到生产环境时，可以关闭颜色输出，以提高性能。

2. 兼容标准库 log 包

一些早期的 Go 项目大量使用了标准库 log 包，如果我们的日志库能够兼容标准库 log 包，我们就可以很容易地替换掉标准库 log 包。例如，logrus 就兼容标准库 log 包。这里，我们来看一个使用了标准库 log 包的代码：

复制代码

```
package main

import (
    "log"
)

func main() {
    log.Print("call Print: line1")
    log.Println("call Println: line2")
}
```

只需要使用`log "github.com/sirupsen/logrus"`替换`"log"`就可以完成标准库 log 包的切换：

复制代码

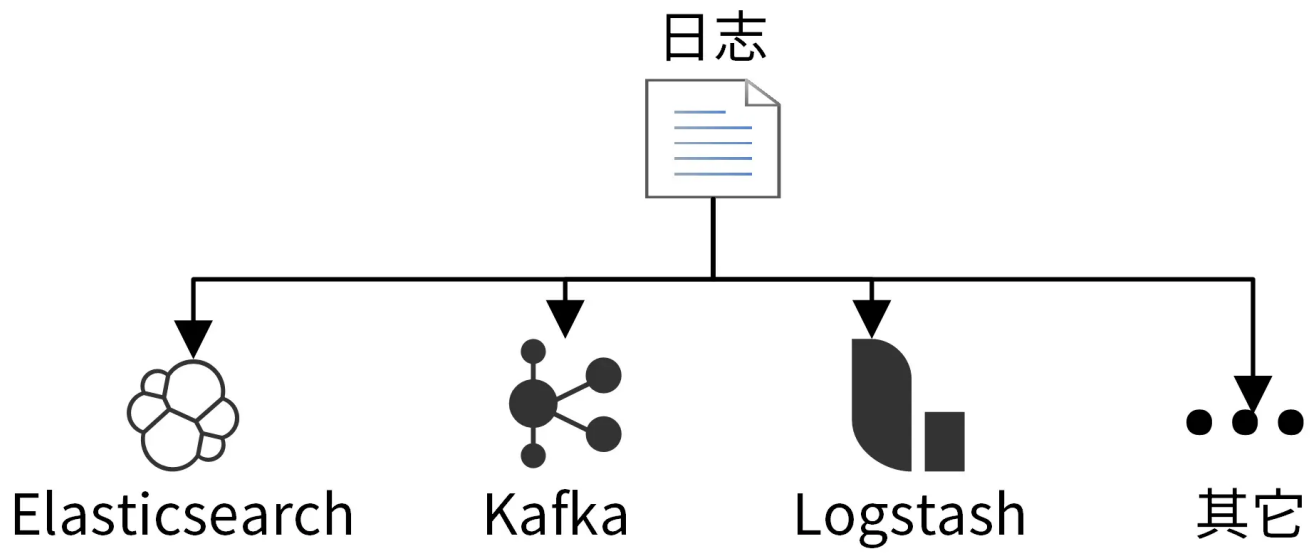
```
package main

import (
    log "github.com/sirupsen/logrus"
)

func main() {
    log.Print("call Print: line1")
    log.Println("call Println: line2")
}
```

3. 支持输出到不同的位置

在分布式系统中，一个服务会被部署在多台机器上，这时候如果我们要查看日志，就需要分别登录不同的机器查看，非常麻烦。我们更希望将日志统一投递到 Elasticsearch 上，在 Elasticsearch 上查看日志。

我们还可能需要从日志中分析某个接口的调用次数、某个用户的请求次数等信息，这就需要我们能够对日志进行处理。一般的做法是将日志投递到 Kafka，数据处理服务消费 Kafka 中保存的日志，从而分析出调用次数等信息。

以上两种场景，分别需要把日志投递到 Elasticsearch、Kafka 等组件，如果我们的日志包支持将日志投递到不同的目的端，那会是一项非常让人期待的功能：



如果日志不支持投递到不同的下游组件，例如 Elasticsearch、Kafka、Fluentd、Logstash 等位置，也可以通过 Filebeat 采集磁盘上的日志文件，进而投递到下游组件。

设计日志包时需要关注的点

上面，我们介绍了日志包具备的功能，这些功能可以指导我们完成日志包设计。这里，我们再来看下设计日志包时，我们还需要关注的几个层面：

高性能：因为我们要在代码中频繁调用日志包，记录日志，所以日志包的性能是首先要考虑的点，一个性能很差的日志包必然会导致整个应用性能很差。

并发安全：Go 应用程序会大量使用 Go 语言的并发特性，也就意味着需要并发地记录日志，这就需要日志包是并发安全的。

插件化能力：日志包应该能提供一些插件化的能力，比如允许开发者自定义输出格式，自定义存储位置，自定义错误发生时的行为（例如 告警、发邮件等）。插件化的能力不是必需的，因为日志自身的特性就能满足绝大部分的使用需求，例如：输出格式支持 JSON 和 TEXT，存储位置支持标准输出和文件，日志监控可以通过一些旁路系统来实现。

日志参数控制：日志包应该能够灵活地进行配置，初始化时配置或者程序运行时配置。例如：初始化配置可以通过 `Init` 函数完成，运行时配置可以通过 `SetOptions` / `SetLevel` 等函数来完成。

如何记录日志？

前面我们介绍了在设计日志包时，要包含的一些功能、实现方法和注意事项。但在这个过程中，还有一项重要工作需要注意，那就是日志记录问题。

日志并不是越多越好，在实际开发中，经常会遇到一大堆无用的日志，却没有我们需要的日志；或者有效的日志被大量无用的日志淹没，查找起来非常困难。

一个优秀的日志包可以协助我们更好地记录、查看和分析日志，但是如何记录日志决定了我们能否获取到有用的信息。日志包是工具，日志记录才是灵魂。这里，我就来详细讲讲如何记录日志。

想要更好地记录日志，我们需要解决以下几个问题：

在何处打印日志？

在哪个日志级别打印日志？

如何记录日志内容？

在何处打印日志？

日志主要是用来定位问题的，所以整体来说，我们要在有需要的地方打印日志。那么具体是哪些地方呢？我给你几个建议。

在分支语句处打印日志。在分支语句处打印日志，可以判断出代码走了哪个分支，有助于判断请求的下一跳，继而继续排查问题。

写操作必须打印日志。写操作最可能会引起比较严重的业务故障，写操作打印日志，可以在出问题时找到关键信息。

在循环中打印日志要慎重。如果循环次数过多，会导致打印大量的日志，严重拖累代码的性能，建议的办法是在循环中记录要点，在循环外面总结打印出来。

在错误产生的最原始位置打印日志。对于嵌套的 Error，可在 Error 产生的最初位置打印 Error 日志，上层如果不需要添加必要的信息，可以直接返回下层的 Error。我给你举个例子：

复制代码

```
package main

import (
    "flag"
    "fmt"

    "github.com/golang/glog"
)

func main() {
    flag.Parse()
    defer glog.Flush()

    if err := loadConfig(); err != nil {
        glog.Error(err)
    }
}

func loadConfig() error {
    return decodeConfig() // 直接返回
}

func decodeConfig() error {
    if err := readConfig(); err != nil {
        return fmt.Errorf("could not decode configuration data for user %s: %v
    }

    return nil
}

func readConfig() error {
    glog.Errorf("read: end of input.")
    return fmt.Errorf("read: end of input")
}
```

通过在最初产生错误的位置打印日志，我们可以很方便地追踪到日志的根源，进而在上层追加一些必要的信息。这可以让我们了解到该错误产生的影响，有助于排障。另外，直接返回下层日志，还可以减少重复的日志打印。

当代码调用第三方包的函数，且第三方包函数出错时，会打印错误信息。比如：

复制代码

```
if err := os.Chdir("/root"); err != nil {
    log.Errorf("change dir failed: %v", err)
}
```

在哪个日志级别打印日志？

不同级别的日志，具有不同的意义，能实现不同的功能，在开发中，我们应该根据目的，在合适的级别记录日志，这里我同样给你一些建议。

1. Debug 级别

为了获取足够的信息进行 Debug，通常会在 Debug 级别打印很多日志。例如，可以打印整个 HTTP 请求的请求 Body 或者响应 Body。

Debug 级别需要打印大量的日志，这会严重拖累程序的性能。并且，Debug 级别的日志，主要是为了能在开发测试阶段更好地 Debug，多是一些不影响现网业务的日志信息。所以，对于 Debug 级别的日志，在服务上线时我们一定要禁止掉。否则，就可能会因为大量的日志导致硬盘空间快速用完，从而造成服务宕机，也可能会影响服务的性能和产品体验。

Debug 这个级别的日志可以随意输出，任何你觉得有助于开发、测试阶段调试的日志，都可以在这个级别打印。

2. Info 级别

Info 级别的日志可以记录一些有用的信息，供以后的运营分析，所以 Info 级别的日志不是越多越好，也不是越少越好，应以满足需求为主要目标。一些关键日志，可以在 Info 级别记录，但如果日志量大、输出频度过高，则要考虑在 Debug 级别记录。

现网的日志级别一般是 Info 级别，为了不使日志文件占满整个磁盘空间，在记录日志时，要注意避免产生过多的 Info 级别的日志。例如，在 for 循环中，就要慎用 Info 级别的日志。

3. Warn 级别

一些警告类的日志可以记录在 Warn 级别，Warn 级别的日志往往说明程序运行异常，不符合预期，但又不影响程序的继续运行，或者是暂时影响，但后续会恢复。像这些日志，就需要你关注起来。Warn 更多的是业务级别的警告日志。

4. Error 级别

Error 级别的日志告诉我们程序执行出错，这些错误肯定会影响到程序的执行结果，例如请求失败、创建资源失败等。要记录每一个发生错误的日志，避免日后排障过程中这些错误被忽略掉。大部分的错误可以归在 Error 级别。

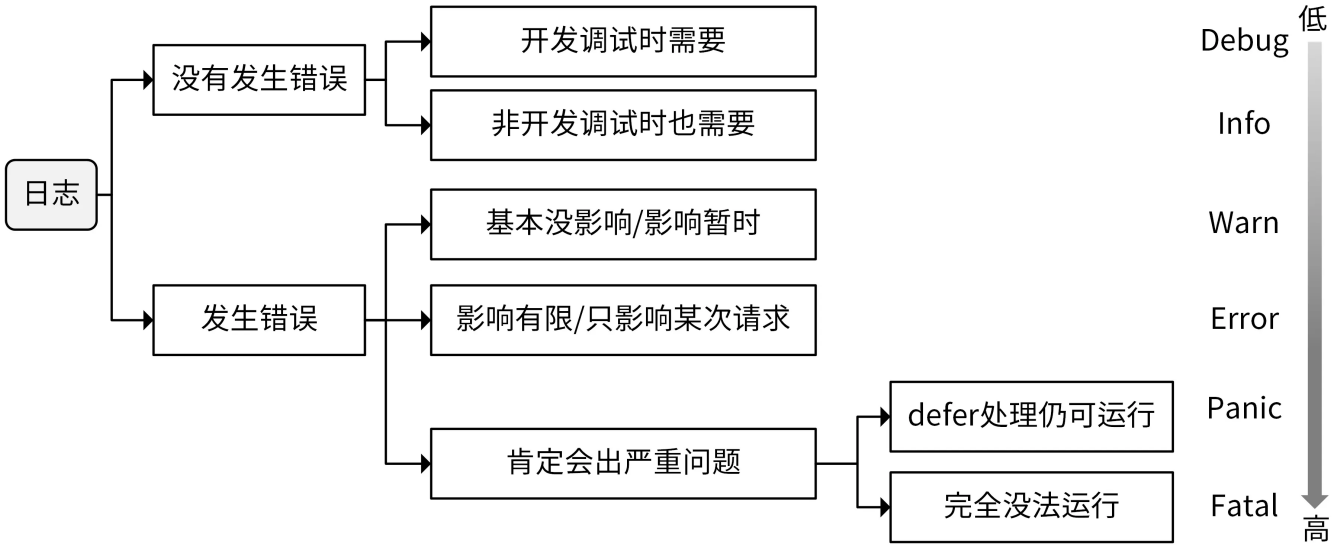
5. Panic 级别

Panic 级别的日志在实际开发中很少用，通常只在需要错误堆栈，或者不想因为发生严重错误导致程序退出，而采用 defer 处理错误时使用。

6. Fatal 级别

Fatal 是最高级别的日志，这个级别的日志说明问题已经相当严重，严重到程序无法继续运行，通常是系统级的错误。在开发中也很少使用，除非我们觉得某个错误发生时，整个程序无法继续运行。

这里用一张图来总结下，如何选择 Debug、Info、Warn、Error、Panic、Fatal 这几种日志级别。



如何记录日志内容？

关于如何记录日志内容，我有几条建议：

在记录日志时，不要输出一些敏感信息，例如密码、密钥等。

为了方便调试，通常会在 Debug 基本记录一些临时日志，这些日志内容可以用一些特殊的字符开头，例如 `log.Debugf("XXXXXXXXXXXX-1:Input key was: %s", setKeyName)`。这样，在完成调试后，可以通过查找 `XXXXXXXXXXXX` 字符串，找到这些临时日志，在 commit 前删除。

日志内容应该小写字母开头，以英文点号 `.` 结尾，例如 `log.Info("update user function called.")`。

为了提高性能，尽可能使用明确的类型，例如使用 `log.Warnf("init datastore: %s", err.Error())` 而非 `log.Warnf("init datastore: %v", err)`。

根据需要，日志最好包含两个信息。一个是请求 ID（RequestID），是每次请求的唯一 ID，便于从海量日志中过滤出某次请求的日志，可以将请求 ID 放在请求的通用日志字段中。另一个是用户和行为，用于标识谁做了什么。

不要将日志记录在错误的日志级别上。例如，我在项目开发中，经常会发现有同事将正常的日志信息打印在 Error 级别，将错误的日志信息打印在 Info 级别。

记录日志的"最佳"实践总结

关于日志记录问题，我从以上三个层面给你讲解了。综合来说，对于日志记录的最佳实践，你在平时都可以注意或进行尝试，我把这些重点放在这里，方便你后续查阅。

开发调试、现网故障排障时，不要遗忘一件事情：根据排障的过程优化日志打印。好的日志，可能不是一次就可以写好的，可以在实际开发测试，还有现网定位问题时，不断优化。但这需要你重视日志，而不是把日志仅仅当成记录信息的一种方式，甚至不知道为什么打印一条 Info 日志。

打印日志要“不多不少”，避免打印没有作用的日志，也不要遗漏关键的日志信息。最好的信息是，仅凭借这些关键的日志就能定位到问题。

支持动态日志输出，方便线上问题定位。

总是将日志记录在本地文件：通过将日志记录在本地文件，可以和日志中心化平台进行解耦，这样当网络不可用，或者日志中心化平台故障时，仍然能够正常的记录日志。

集中化日志存储处理：因为应用可能包含多个服务，一个服务包含多个实例，为了查看日志方便，最好将这些日志统一存储在同一个日志平台上，例如 Elasticsearch，方便集中管理和查看日志。

结构化日志记录：添加一些默认通用的字段到每行日志，方便日志查询和分析。

支持 RequestID：使用 RequestID 串联一次请求的所有日志，这些日志可能分布在不同的组件，不同的机器上。支持 RequestID 可以大大提高排障的效率，降低排障难度。在一些大型分布式系统中，没有 RequestID 排障简直就是灾难。

支持动态开关 Debug 日志：对于定位一些隐藏得比较深的问题，可能需要更多的信息，这时候可能需要打印 Debug 日志。但现网的日志级别会设置为 Info 级别，为了获取 Debug 日志，我们可能会修改日志级别为 Debug 级别并重启服务，定位完问题后，再修改日志级别为 Info 级别，然后再重启服务，这种方式不仅麻烦而且还可能会对现网业务造成影响，最好的办法是能够在请求中通过 `debug=true` 这类参数动态控制某次请求是否开启 Debug 日志。

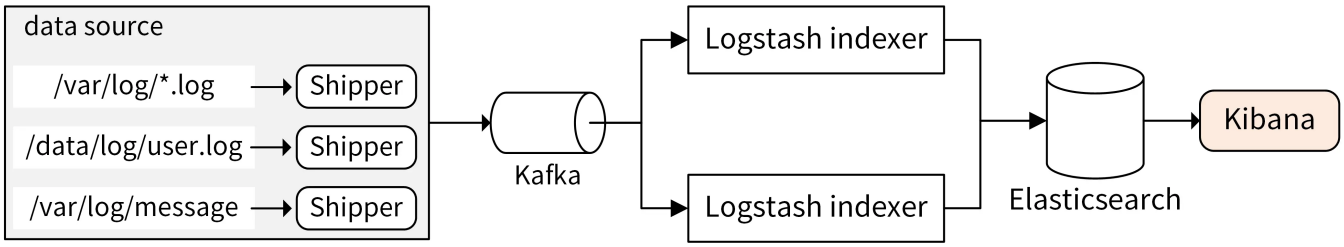
拓展内容：分布式日志解决方案（EFK/ELK）

前面我们介绍了设计日志包和记录日志的规范，除此之外，还有一个问题你应该了解，那就是：我们记录的日志如何收集、处理和展示。

在实际 Go 项目开发中，为了实现高可用，同一个服务至少需要部署两个实例，通过轮询的负载均衡策略转发请求。另外，一个应用又可能包含多个服务。假设我们的应用包含两个服务，每个服务部署两个实例，如果应用出故障，我们可能需要登陆 4（2 x 2）台服务

器查看本地的日志文件，定位问题，非常麻烦，会增加故障恢复时间。所以在真实的企业场景中，我们会将这些日志统一收集并展示。

在业界，日志的收集、处理和展示，早已经有了一套十分流行的日志解决方案：EFK（Elasticsearch + Filebeat + Kibana）或者 ELK（Elasticsearch + Logstash + Kibana），EFK 可以理解为 ELK 的演进版，把日志收集组件从 Logstash 替换成了 Filebeat。用 Filebeat 替换 Logstash，主要原因是 Filebeat 更轻量级，占用的资源更少。关于日志处理架构，你可以参考这张图。



通过 log 包将日志记录在本地文件中（*.log 文件），再通过 Shipper 收集到 Kafka 中。Shipper 可以根据需要灵活选择，常见的 Shipper 有 Logstash Shipper、Flume、Fluentd、Filebeat。其中 Filebeat 和 Logstash Shipper 用得最多。Shipper 没有直接将日志投递到 Logstash indexer，或者 Elasticsearch，是因为 Kafka 能够支持更大的吞吐量，起到削峰填谷的作用。

Kafka 中的日志消息会被 Logstash indexer 消费，处理后投递到 Elasticsearch 中存储起来。Elasticsearch 是实时全文搜索和分析引擎，提供搜集、分析、存储数据三大功能。Elasticsearch 中存储的日志，可以通过 Kibana 提供的图形界面来展示。Kibana 是一个基于 Web 的图形界面，用于搜索、分析和可视化存储在 Elasticsearch 中的日志数据。

Logstash 负责采集、转换和过滤日志。它支持几乎任何类型的日志，包括系统日志、错误日志和自定义应用程序日志。Logstash 又分为 Logstash Shipper 和 Logstash indexer。其中，Logstash Shipper 监控并收集日志，并将日志内容发送到 Logstash indexer，然后 Logstash indexer 过滤日志，并将日志提交给 Elasticsearch。

总结

记录日志，是应用程序必备的功能。记录日志最大的作用是排障，如果想更好地排障，我们需要一个优秀的工具，日志包。那么如何设计日志包呢？首先我们需要知道日志包的功能，在我看来日志包需要具备以下功能：

基础功能：支持基本的日志信息、支持不同的日志级别、支持自定义配置、支持输出到标准输出和文件。

高级功能：支持多种日志格式、能够按级别分类输出、支持结构化日志、支持日志轮转、具备 Hook 能力。

可选功能：支持颜色输出、兼容标准库 log 包、支持输出到不同的位置。

另外，一个日志包还需要支持不同级别的日志，按日志级别优先级从低到高分别是：Trace < Debug < Info < Warn/Warning < Error < Panic < Fatal。其中 Debug、Info、Warn、Error、Fatal 是比较基础的级别，建议在开发一个日志包时包含这些级别。Trace、Panic 是可选的级别。

在我们掌握了日志包的功能之后，就可以设计、开发日志包了。但我们在开发过程中，还需要确保我们的日志包具有比较高的性能、并发安全、支持插件化的能力，并支持日志参数控制。

有了日志包，我们还需要知道如何更好地使用日志包，也就是如何记录日志。在文中，我给出了一些记录建议，内容比较多，你可以返回文中查看。

最后，我还给出了分布式日志解决方案：EFK/ELK。EFK 是 ELK 的升级版，在实际项目开发中，我们可以直接选择 EFK。在 EFK 方案中，通过 Filebeat 将日志上传到 Kafka，Logstash indexer 消费 Kafka 中的日志，并投递到 Elasticsearch 中存储起来，最后通过 Kibana 图形界面来查看日志。

课后练习

思考一下，你的项目中，日志包还需要哪些功能，如何设计？你的日常开发中，如果有比较好的日志记录规范，也欢迎在留言区分享讨论。

期待你的思考和看法，我们下一讲见。

分享给需要的人，Ta订阅后你可得 24 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 错误处理（下）：如何设计错误包？

下一篇 21 | 日志处理（下）：手把手教你从 0 编写一个日志包

精选留言 (3)

写留言

helloworld

2021-07-10

“日志内容应该小写字母开头，以英文点号 . 结尾” ，如果日志结尾有符号，在代码静态检查时，提示日志内容结尾不应有符号的，这块和您说的不一致

展开

作者回复: 可以不以 . 号结尾，其实影响不大，形成规范即可。

2

XI

小哥哥 2021-07-13

请问下老师，大型的分布式服务都需要日志的链路追踪，日志的trance_id 由网关生成，每个微服务在接受请求的时候也应当必传trance_id 作为请求记录，大公司是这样设计的吗？

作者回复: 大公司也是这么设计的。

timidsmile

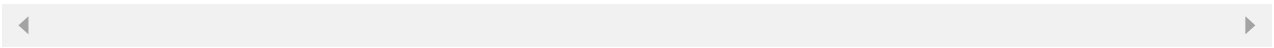
2021-07-12

对于日志轮转功能，其实我不建议在日志包中添加

这里使用Logrotate的话，是使用logrotate的哪种方式呢？
* copytruncate 方式的话，对于大日志应该有性能问题吧？
* create 方式的话，是需要通过接收特殊signal的方式来实现重新打开日志吗？考虑到很...

展开

作者回复: copytruncate比较好吧，inode不变，这样应用程序可以继续写文件，也比较简单。一般一个系统一天的日志50G算是比较大的了，copy 50G的日志，不会给服务器带来压力的。



1